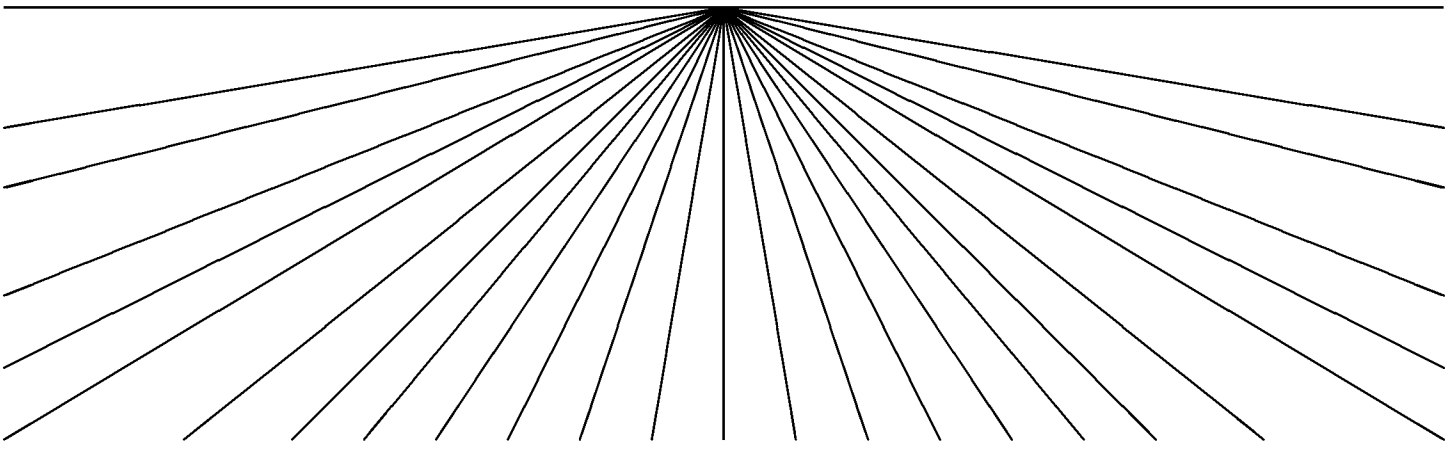
**facultad de informática**

universidad politécnica de madrid

# Towards Extracting Non–Strict Independent And–Parallelism Using Sharing and Freeness Information

Daniel Cabeza Gras
Manuel Hermenegildo

**TR Number** CLIP 5/92.1(93)



# Towards Extracting Non–Strict Independent And–Parallelism Using Sharing and Freeness Information

Technical Report Number: CLIP 5/92.1(93)

**April 1993**

Authors

**Daniel Cabeza Gras**
dcabeza@dia.fi.upm.es
**Manuel Hermenegildo**
herme@fi.upm.es

Departamento de Inteligencia Artificial
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid - SPAIN

Keywords

Parallel Execution of Logic Programs, Generation of Annotations for Parallelism, Abstract Interpretation, Non–strictly Independent And–Parallelism.

Abstract

Logic programming systems which exploit and–parallelism among non–deterministic goals rely on notions of independence among those goals in order to ensure certain efficiency properties. "Non–strict" independence (NSI) is a more relaxed notion than the traditional notion of "strict" independence (SI) which still ensures the relevant efficiency properties and can allow considerable more parallelism than SI. However, all compilation technology developed to date has been based on SI, presumably because of the intrinsic complexity of exploiting NSI. This is related to the fact that NSI cannot be determined "a priori" as SI. This paper fills this gap by developing a technique for compile–time detection and annotation of NSI. It also proposes algorithms for combined compile–time/run–time detection, presenting novel run-time checks for this type of parallelism. Also, a transformation procedure to eliminate shared variables among parallel goals is presented, attempting to perform as much work as possible at compile-time. The approach is based on the knowledge of certain properties about run-time instantiations of program variables —sharing and freeness— for which compile–time technology is available, with new approaches being currently proposed.

## Contents

## 1 Introduction

Several types of parallel logic programming systems and models exploit and–parallelism [4] among non–deterministic goals. Some examples are PEPsys [27], ROPM [20], AO-WAM [8], DDAS/Prometheus [22], systems based on the "Extended" Andorra Model [26] such as AKL [14], and &-Prolog [10] (please see their references for other related systems). All these systems rely on some notion of independence (or the related notion of "stability" [9]) among non–deterministic goals being run in and–parallel in order to ensure certain important efficiency properties. Two basic notions of independence defined so far are strict and non–strict independence [11, 12].

Strict independence corresponds to the traditional notion of independence among goals [4, 7, 10]: Two goals $g_1$ and $g_2$ are said to be strictly independent for a substitution $\theta$ iff $\text{var}(g_1\theta) \cap \text{var}(g_2\theta) = \emptyset$; $n$ goals $g_1, \ldots, g_n$ are said to be strictly independent for a substitution $\theta$ if they are pairwise strictly independent for $\theta$. Parallelization of strictly independent goals has the property of preserving the search space of the goals involved so that correctness and efficiency of the original program are maintained and a no speed–down condition can be ensured [11]. A convenient characteristic of strict independence is that it is an "a priori" condition, i.e. it can be tested at run–time ahead of the execution of the goals. Furthermore, tests for strict independence can be expressed directly in terms of groundness and independence of the variables involved. This allows relatively simple compile–time parallelization by introducing run–time tests in the program [7, 17]. These tests can then be partially eliminated at compile–time by direct application of groundness and sharing (independence) information obtained from global analysis [19].

Non–strict independence is a relaxation of strict independence traditionally defined as follows: given a collection of goals $g_1, \ldots, g_n$ and a substitution $\theta$, let $SH = \{v \mid \exists i,j\ 1 \leq i < j \leq n, v \in \text{var}(g_i\theta) \cap \text{var}(g_j\theta)\}$, let $\theta_i$ be any answer substitution for $g_i\theta$, then $g_1, \ldots, g_n$ are non–strictly independent for $\theta$ iff $\forall v \in SH$, at most the rightmost $g_i$ such that $v \in \text{var}(g_i\theta)$ binds $v$ to a non–variable term, and if $\text{var}(g_i\theta)$ contains more than one variable of $SH$, say $x_1, \ldots, x_k$, then $x_1\theta_i, \ldots, x_k\theta_i$ are *strictly independent* [12]. Non–strict independence is clearly a more powerful notion than strict independence since strictly independent goals are always non–strictly independent. Furthermore, it still preserves the same properties as strict independence with respect to correctness and efficiency, and do not alter the left to right sequential computation semantics of Prolog. In practice, it has wide application for example in the parallelization of programs which

use difference lists and incomplete structures in general. In fact, studies of amounts of ideal parallelism in logic programs suggest that there is a potential for large speedups from the exploitation of non–strict independence [22]. However, this potential remains untapped from the point of view of automatic parallelization. This is due to two factors. The first one is that non–strict independence is not an "a priori" condition, i.e. it cannot be expressed simply in terms of run–time tests (without running the goals). Thus, run–time detection by itself is ruled out. Unfortunately, compile–time detection is complicated by the fact that non–strict independence is not directly expressed in the same terms as the properties which are usually determined from global analysis.

Earlier studies [11] have suggested that coupling sharing and groundness analysis with freeness analysis could be instrumental in the task of non–strict independence detection. This has been one of the motivations behind the development of analyzers capable of inferring these three types of information [3, 5, 24, 18, 25]. However, there still remained a semantic gap between the availability of that information and actually being able to reason about the non–strict independence of a set of goals. This paper attempts to fill this gap. It aims to develop concrete techniques for determining non–strict independence at compile–time. For concreteness, it focuses on a concrete way of expressing sharing and freeness information, the sharing+freeness domain [18]. This allows a high degree of precision in the conditions involved, which are given in such a way that the implementation is straightforward. However, we believe that the ideas presented can also be used for related domains, provided that these domains give information about variable sharing and freeness.

One design decision throughout the paper is to concentrate on the parallelization of two goals or sets of goals (containing either sequential or parallel constructs). This is convenient from a practical point of view because many parallelization algorithms work by repeatedly considering whether a pair of goals or sequences are independent while, for example, building a dependency graph [17]. The algorithms described in this paper are directly aimed at answering such questions for the case of non–strict independence. The decision of considering the parallelization of pairs of goals has also a sound theoretical foundation. Consider the following alternative definition of non–strict independence: Given two goals $g_1$ and $g_2$, where $g_2$ is to the right of $g_1$, and a substitution $\theta$, consider the set of shared variables $SH = \mathrm{var}(g_1\theta) \cap \mathrm{var}(g_2\theta)$. Then, $g_1$ and $g_2$ are non–strictly independent for $\theta$ iff for any answer substitution $\theta_1$ of $g_1\theta$ and for all $v$,$w \in SH$, $v\theta_1$ is a variable and $v \neq w \rightarrow v\theta_1 \neq w\theta_1$. Based on this, the definition involving $n$ goals can be expressed as: $g_1, \ldots, g_n$ are non–strictly independent for a substitution $\theta$ if they are pairwise non–strictly independent for $\theta$. Clearly, this is equivalent to the standard definition, and thus considering only a pair of goals can

always be done without loss of generality.

The rest of the paper proceeds as follows: Section 2 introduces the concept of Abstract Interpretation, on which the technique proposed is based. Section 3 explains the particular domain for which the conditions of parallelism are given, the sharing+freeness domain, and introduces a pictorial representation for the abstract substitutions involved. Section 4 presents, by a refinement process, the sufficient conditions found for compile–time detection of NSI. Section 5 deals with the combination of compile–time analyses and run–time checks for detecting NSI, presenting novel run–time checks for this type of parallelism. It also connects this method with the previously proposed techniques for the detection of strict independence. Section 6 develops an efficient algorithm for performing combined compile–time/run–time renaming of variables, which is needed for the parallel execution of non–strictly independent goals. Section 7 shows how the techniques proposed can be used for the parallelization of a concrete program, and why non–strict independence is more powerful than strict independence. Section 8 proposes new approaches related to compile–time analysis in order to improve the information required for the parallelization techniques. Finally, section 9 gives the conclusions and points out future work.

## 2 Abstract Interpretation of Logic Programs

Although a detailed introduction to abstract interpretation is outside the scope of this paper, this section briefly presents a minimal background (see [6] for details). As mentioned previously, abstract interpretation is a useful technique for performing a global analysis of a program in order to compute, at compile–time, characteristics of the terms to which the variables in that program will be bound at run–time (for a given class of queries).

Abstract interpretation uses the notions of *approximation* and *finite representation* to make the problem of compile–time analysis tractable. Approximation is based on the observation that if we construct a set $\Theta_a \supseteq \Theta$, and prove that $\forall\, \theta \in \Theta_a \;\; p(\theta)$, then the property holds also for $\Theta$. $\Theta_a$ is said to be a *safe approximation* of $\Theta$. Any function (such as unification, for example) can also be approximated in a similar way.

The second basic concept is that of *finite representation.* Given a semantic function $F_P^*$ describing the meaning of a program over a domain $\wp(D)$ of sets of concrete values, $\wp(D)$ can be represented by an "abstract" domain $D_\alpha$ whose elements are finite representations of (possibly) infinite objects in $\wp(D)$. Thus, in the case of analyzing substitutions at some point in a clause, the concrete domain $D$ is the set of all substitu-

tions for the variables which can appear in that point clause. The abstract domain $D_\alpha$ is then the set of all "abstract substitutions", an abstract substitution $\lambda$ being a finite representation of a, possibly infinite, set of actual substitutions. The representation of $\wp(D)$ by $D_\alpha$ is expressed by a (monotonic) function called a *concretization function*: $\gamma\colon D_\alpha \to \wp(D)$ such that $\gamma(\lambda) = d$ iff $d$ is the largest element (under $\subseteq$) of $\wp(D)$ that $\lambda$ describes. Note that $(\wp(D), \subseteq)$ is obviously a complete lattice. We can also define a (monotonic) *abstraction function* $\alpha\colon D \to D_\alpha$, where $\alpha(d) = \lambda$ iff $\lambda$ is the "least" element of $D_\alpha$ that describes $d$.

An *abstract semantic function* can then be defined as $F_\alpha\colon D_\alpha \to D_\alpha$ which is is a safe approximation of the standard semantic function if $\forall\, \lambda \in D_\alpha \;\; \gamma(F_\alpha(\lambda)) \supseteq F_P^*(\gamma(\lambda))$. It is then possible to prove a property of the output of a given class of inputs represented by $\lambda$ by proving that all elements of $\gamma(F_\alpha(\lambda))$ have such property.

## 3 Understanding Sharing+Freeness Abstract Substitutions

The sharing+freeness abstract domain [18] (other related analyses for which our results may be valid include [3, 5, 24, 25]) was proposed with the objective of obtaining at compile–time accurate variable *groundness*, *sharing*, and *freeness* information for a program, i.e., respectively, information on when a program variable will be bound to a ground term, when a set of program variables will be bound to terms with variables in common, and when a program variable will be unbound or bound only to other variables instead of to a complex term.

The abstract domain approximates this information by combining two components (in fact domains per se): the first component provides information on sharing (aliasing, independence) and groundness [13, 16]; the second one provides information on freeness. More precisely, $D_\alpha \subset \bot \cup \wp(\wp(Pvar)) \times \wp(Pvar)$, where $Pvar$ is the set of all program variables in the current clause. It is an inclusion and not an equality because abstract substitutions in $\wp(\wp(Pvar)) \times \wp(Pvar)$ whose concretization would be empty are not considered (they are represented by $\bot$ —bottom).

We will denote a sharing+freeness abstract substitution as a pair (sharing, freeness) as in $\widehat{\theta} = (\widehat{\theta}_{\mathrm{SH}}, \widehat{\theta}_{\mathrm{FR}})$. To distinguish abstract substitutions from concrete substitutions abstract substitutions will be represented by greek letters with a hat, the same greek letter without hat representing a concrete substitution approximated by the abstract one. Sets will be denoted with square brackets in abstract substitutions (to distinguish them and because of the mnemonic connotations since they are to be represented in Prolog in the analyzer), and with braces in concrete substitutions (as usual).

Informally, an abstract substitution in the sharing domain is a set of sets of program variables (a set of sharing sets), where sharing sets represent all possible sharing patterns among the program variables.

More formally, let us define a (concrete) substitution in a clause as a mapping from the set of program variables in that clause (*Pvar*) to terms that can be formed from the constants and the functors in the given program and in the query and an infinite set of variables *Var* such that $Pvar \cap Var = \emptyset$. In this way we consider only idempotent substitutions. The set of all concrete substitutions will be denoted as *Subst*.

The function $Occ$: $Subst \times Var \rightarrow \wp(Pvar)$ is defined such that

$$Occ(\theta, V) = \{X \in \mathrm{dom}(\theta) \mid V \in \mathrm{var}(X\theta)\}$$

where $t\theta$ denotes the instantiation of a term $t$ under a substitution $\theta$, $\mathrm{var}(t\theta)$ denotes the set of all variables in $t\theta$ and $\mathrm{dom}(\theta)$ denotes the domain of a substitution $\theta$. In other words, the function returns the set of all program variables $X$ such that $V$ occurs in the instantiation of $X$ under $\theta$. The abstraction of a substitution $\theta$ in the sharing domain is defined as:

$$\alpha_{\mathrm{SH}}(\theta) = \{Occ(\theta, V) \mid V \in \mathrm{range}(\theta)\}$$

The concretization of an abstract substitution in the sharing domain is defined as

$$\gamma(\widehat{\theta}_{\mathrm{SH}}) = \{\theta \in Subst \mid \alpha_{\mathrm{SH}}(\theta) \subseteq \widehat{\theta}_{\mathrm{SH}}\}$$

For example, given the following concrete substitution $\theta$, $\widehat{\theta}_{\mathrm{SH}}$ is its abstraction in the sharing domain:

$$\begin{array}{lcl} \theta & = & \{\mathrm{X/f(1,a)},\ \mathrm{Y/A},\ \mathrm{Z/f(A,C,t(B))},\ \mathrm{W/[B,C]},\ \mathrm{V/D}\} \\ \widehat{\theta}_{\mathrm{SH}} & = & [[\mathrm{YZ}]\,[\mathrm{ZW}]\,[\mathrm{V}]] \end{array}$$

On the other hand, given the following sharing abstract substitution $\widehat{\theta}_{\mathrm{SH}}$, the $\theta_i$ are concrete substitutions approximated by it. The last column in the following represents the sharing sets "active" in each concrete substitution –we say that a set $L \in \widehat{\theta}_{\mathrm{SH}}$, where $\widehat{\theta}_{\mathrm{SH}}$ is a sharing abstract substitution, is **active** in a concrete substitution $\theta \in \gamma(\widehat{\theta}_{\mathrm{SH}})$ iff $L \in \alpha(\theta)$, i.e. $L$ is in the abstraction of $\theta$:

$$
\begin{array}{lcll}
\widehat{\theta}_{\mathrm{SH}} & = & [[\mathrm{X}]\,[\mathrm{YZ}]\,[\mathrm{ZW}]] & \\
\theta_1 & = & \{\mathrm{X/A},\ \mathrm{Y/f(B,1)},\ \mathrm{Z/B},\ \mathrm{W/foo}\} & [[\mathrm{X}]\,[\mathrm{YZ}]] \\
\theta_2 & = & \{\mathrm{X/[\,]},\ \mathrm{Y/A},\ \mathrm{Z/[B|A]},\ \mathrm{W/t(B)}\} & [[\mathrm{YZ}]\,[\mathrm{ZW}]] \\
\theta_3 & = & \{\mathrm{X/t(0,1)},\ \mathrm{Y/atom},\ \mathrm{Z/A},\ \mathrm{W/A}\} & [[\mathrm{ZW}]]
\end{array}
$$

The component described above, is essentially the abstract domain of Jacobs and Langen [13].

An abstract substitution in the freeness domain is a set of program variables (those that are known to be free). More formally, the abstraction and concretization functions in this domain are defined as follows:

$$
\begin{array}{lcl}
\alpha_{\mathrm{FR}}(\theta) & = & \{X \in \mathrm{dom}(\theta) \mid X\theta \in \mathit{Var}\} \\
\gamma(\widehat{\theta}_{\mathrm{FR}}) & = & \{\theta \in \mathit{Subst} \mid \alpha_{\mathrm{FR}}(\theta) \supseteq \widehat{\theta}_{\mathrm{FR}}\}
\end{array}
$$

The concretization of a sharing+freeness abstract substitution can be defined as the intersection of the concretizations of its two components:

$$\gamma(\widehat{\theta}) = \gamma(\widehat{\theta}_{\mathrm{SH}}) \cap \gamma(\widehat{\theta}_{\mathrm{FR}})$$

The set inclusion relation in the concrete domain induces a *partial order* on the abstract substitutions, i.e. $\widehat{\phi} \sqsubseteq \widehat{\psi}$ iff $\gamma(\widehat{\phi}) \subseteq \gamma(\widehat{\psi})$. The function *lub* computes the least upper bound of two abstract substitutions $\widehat{\phi}$ and $\widehat{\psi}$ by taking the least upper bound of their *sharing* and *freeness* components:

$$\mathit{lub}(\widehat{\phi}, \widehat{\psi}) = (\widehat{\phi}_{\mathrm{SH}} \cup \widehat{\psi}_{\mathrm{SH}}, \widehat{\phi}_{\mathrm{FR}} \cap \widehat{\psi}_{\mathrm{FR}})$$

It is important to point out that the approximations performed by the abstraction function and the *lub* function with respect to the sharing component imply that this component can actually represent in a compact way (rather than an explicit disjunction) several combinations of sharing patterns. One of the main sources of information in being able to tell these combinations apart is the freeness information. In fact, sharing information is not independent of freeness information since known freeness of certain variables restricts the allowable combination of sharing patterns. The possible combinations of sharing sets a sharing+freeness abstract substitution $\widehat{\theta}$ represents are the subsets of the sharing component (the $S \in \wp(\widehat{\theta}_{\mathrm{SH}})$) that have one and only one sharing set including each variable in the freeness component ($\forall v \in \widehat{\theta}_{\mathrm{FR}}\, \dot{\exists}\, L \in S\ v \in L$).

The point above regarding sharing+freeness abstract substitutions, which is of great practical importance, may still be difficult to understand in the terms given so far. It is hoped that with the aid of the pictorial representation to be presented in the following section these issues will be greatly clarified.

### 3.1 Pictorial Representation of Substitutions

We have chosen a pictorial representation of substitutions in order to make it easier to understand abstract substitutions in the sharing+freeness domain and to follow the discussions and examples throughout the text. The idea of the pictures is to make the large amount of information contained in these abstract substitutions explicit. Figure 1 illustrates the different types of objects used in this representation.

As mentioned before, an abstract sharing+freeness substitution is a compact representation of a finite number of possible sharing+freeness situations in the concrete domain. To reflect this a given sharing+freeness abstract substitution can be represented with a finite number of figures, each figure having the same freeness information (which is definite) but representing an alternative valid combination of sharing sets (i.e. a subset of the sharing component).

Variables in the freeness component are represented with dots, the rest with circles. The sharing patterns are represented with connected lines going to each variable of the corresponding sharing set. The resulting pictures are hypergraphs, i.e. graphs where the edges connect an arbitrary number of vertices.

Thus, the number of edges connected to a vertex is the number of sharing sets containing the corresponding variable, except for dot vertices (free variables) that can have multiple edges, all corresponding to the same sharing pattern; or none, meaning a sharing pattern with only this variable (since free variables must be in one and only one sharing pattern). A ground variable appears like an isolated circle.

A goal is represented like a set in a Venn Diagram, the variables in the set being the goal variables. When we represent two goals, the first one is to the left and the second one to the right, and the variables present in both goals are put in the intersection. We will print an arrow from a picture to another picture whenever there can be a transition from the first situation to the second. The same arrow crossed with a slash means that the transition is impossible.

Figure 2 shows an example that represents an abstract substitution with only one picture, corresponding to a unique possible combination of the sharing sets. In it, W is

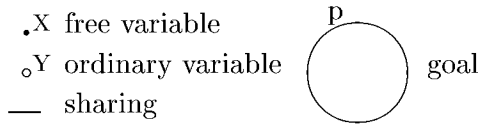



Figure 1: Types of objects in our pictorial representation.

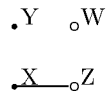



Figure 2: Variables X, Y, Z, W with $\widehat{\theta}$ = ([[Y] [XZ]], [XY])

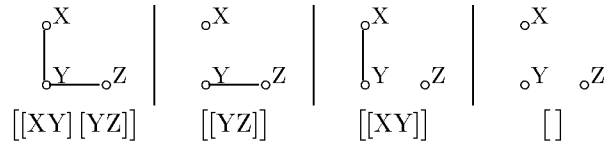



Figure 3: Variables X, Y, Z with $\widehat{\theta}$ = ([[XY] [YZ]], [ ])

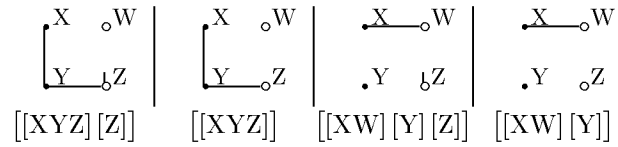



Figure 4: Variables X, Y, Z, W with $\widehat{\theta}$ = ([[XYZ] [XW] [Y] [Z]], [XY])

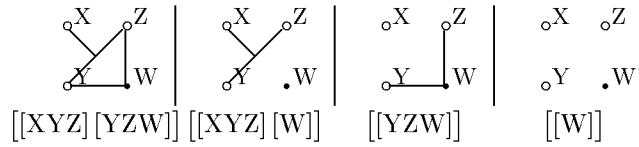



Figure 5: Variables X, Y, Z, W with $\widehat{\theta}$ = ([[XYZ] [YZW] [W]], [W])

ground (because it does not appear in any sharing set), Y is free and not shared, and Z is a term that contains the free variable X.

Figures 3 to 5 show other examples that represent more complex abstract substitutions with several pictures. The particular subset of the sharing component each picture represents is displayed below it.

Figure 3 gives an example where there are no variables in the freeness component, so all combinations of sharing sets are possible. On the other hand, the next figure shows an abstract substitution with non–empty freeness component, which restricts the combination of sharing sets: [Z] can be active or not, but [XYZ] on the one hand and [XW] together with [Y] on the other are mutually exclusive. Finally, figure 5 illustrates the different treatment of sharing sets depending on whether they have free variables or not: [XYZ] is drawn like a hyper-edge because it has no variables in the freeness component, on the other hand, [YZW] appears like two edges connecting Y and Z to the free variable W, showing that this and no other variable is shared among them.

# 4 Conditions for Non–Strict Independence with Respect to the Information from Sharing+Freeness Analysis

As mentioned in the introduction we will consider the parallelization of pairs of goals. Let $\tilde{p}$ and $\tilde{q}$ be two goals or sequences of goals, where $\tilde{q}$ is to the right of $\tilde{p}$. Also let $\widehat{\beta}$ and $\widehat{\psi}$ be the call and answer abstract substitutions for $\tilde{p}$. We define the following sets:

$$\begin{array}{lcl} \mathrm{SH}(\tilde{g}) & = & \{L \in \widehat{\beta}_{\mathrm{SH}} \mid \exists\, v \in \mathrm{var}(\tilde{g})\ \ v \in L\} \\ \mathrm{SH} & = & \mathrm{SH}(\tilde{p}) \cap \mathrm{SH}(\tilde{q}) \\ \mathrm{FR}(L) & = & L \cap \widehat{\beta}_{\mathrm{FR}} \\ \mathrm{FR} & = & \bigcup_{L \in \mathrm{SH}} \mathrm{FR}(L) \end{array}$$

The sharing sets of $\mathrm{SH}(\tilde{g})$ contain all the variables accessible by $\tilde{g}$ under the abstract substitution $\widehat{\beta}$ (i.e. accessible under one concrete substitution represented by $\widehat{\beta}$). Then, SH contains all the variables that $\tilde{p}$ can modify and at the same time can affect the execution of $\tilde{q}$. On the other hand, if a variable in $\widehat{\beta}_{\mathrm{FR}}$ appears in the sharing set $L$, this is the variable shared in that set (if active), so the variables in $\mathrm{FR}(L)$ access at runtime to the same free variable.

## 4.1 Compile–Time Conditions. First Approach

The following are sufficient conditions to execute $\tilde{p}$ and $\tilde{q}$ in parallel, provided that no transitive dependencies between them occur (i.e. the conditions guarantee non–strict independence of the goals):

$$\begin{array}{ll} \mathrm{C1}_1 & \forall\, L \in \mathrm{SH}\ \ \mathrm{FR}(L) \neq \emptyset \\ \mathrm{C2}_1 & \mathrm{FR} \subseteq \widehat{\psi}_{\mathrm{FR}} \\ \mathrm{C3}_1 & \neg\,(\exists\, L \in \widehat{\psi}_{\mathrm{SH}}\, \exists\, L_1,L_2 \in \mathrm{SH}\, \exists\, v_1,v_2 \in \mathrm{FR} \\ & \quad v_1,v_2 \in L \wedge v_1 \in L_1 \wedge v_1 \notin L_2 \wedge v_2 \in L_2 \wedge v_2 \notin L_1) \end{array}$$

Condition $\mathrm{C1}_1$ says that every run–time free variable shared by $\tilde{p}$ and $\tilde{q}$ must be in $\widehat{\beta}_{\mathrm{FR}}$, in order to verify that it remains free in the answer abstract substitution $\widehat{\psi}$ (condition $\mathrm{C2}_1$). Condition $\mathrm{C3}_1$ deals with preserving independence of shared variables: it says that any two free variables shared by $\tilde{p}$ and $\tilde{q}$ are not allowed to be together in a set of $\widehat{\psi}_{\mathrm{SH}}$ and at the same time each one appear in a set of SH in which the other one does not occur (because the $\tilde{p}$ execution possibly makes them dependent).

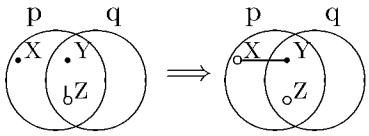



Figure 6: A situation without NSI where parallelism is correctly avoided by condition $C1_1$.

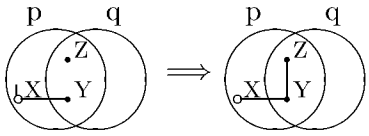



Figure 7: A situation without NSI where parallelism is correctly avoided by condition $C3_1$.

Figure 6 depicts one situation where the conditions do not hold, due to the fact that the goals may not be non–strictly independent. Here, the call and answer abstract substitutions for p̃ are $\widehat{\beta}$ = ([[X] [Y] [Z]], [XY]) and $\widehat{\psi}$ = ([[XY] [Z]], [Y]). The first condition is not fulfilled: [Z] $\in$ SH but FR([Z]) = $\emptyset$. A concrete situation lacking non–strict independence would be, for example, if $\beta$ = {X/A, Y/B, Z/f(C)} and $\psi$ = {X/[1, 2|B], Y/B, Z/f(3)} (note that the binding of X yields no problems, since X is not shared).

Figure 7 gives an example where the third condition does not hold. The call and answer abstract substitutions are $\widehat{\beta}$ = ([[X] [XY] [Z]], [YZ]) and $\widehat{\psi}$ = ([[XYZ]], [YZ]). In this situation two shared variables are unified; we have $v_1$ = Y, $v_2$ = Z, $L$ = [XYZ], $L_1$ = [XY] and $L_2$ = [Z]. An example in terms of concrete substitutions is $\beta$ = {X/f(C, A), Y/A, Z/B} and $\psi$ = {X/f(foo, A), Y/A, Z/A}.

Nevertheless, there are still some cases that do not meet condition $C3_1$ but can be parallelized, since it is not possible to go from an independent situation under $\widehat{\beta}$ to a dependent one under $\widehat{\psi}$. For example, in figure 8, we have the call and answer abstract substitutions $\widehat{\beta} = \widehat{\psi}$ = ([[X] [YZ] [XY]], [XY]), with $v_1$ = X, $v_2$ = Y, $L$ = [XY], $L_1$ = [X] and $L_2$ = [YZ]. The variable Z, that depends on Y in $L_2$, is not present in the sharing set $L$, and it should be present.

Figure 9 shows another situation that can be parallelized but in which $C3_1$ prevents doing so; we have $\widehat{\beta} = \widehat{\psi}$ = ([[X] [Y] [XYZ]], [XY]), $v_1$ = X, $v_2$ = Y, $L$ = [XYZ], $L_1$ = [X] and $L_2$ = [Y]. The variable Z in the sharing set $L$ does not appear in any other set of SH, so p̃ cannot possibly make all the variables in $L$ share. Note that these cases arise when the abstract substitutions represent alternative combinations of sharing patterns.

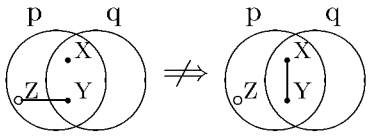



Figure 8: Impossible transition: Z should be in [XY].

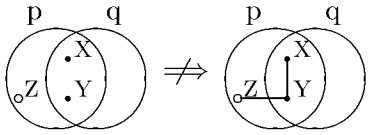



Figure 9: Impossible transition: Z in [XYZ] not in other set of SH.

In the next section we will provide a definition which captures the cases now missed.

### 4.2 Compile–Time Conditions. Final Approach

As we have seen, to exploit more parallelism we must relax the condition $C3_1$. The following are our proposed conditions:

C1 $\forall\, L \in \mathrm{SH}\ \ \mathrm{FR}(L) \neq \emptyset$

C2 $\mathrm{FR} \subseteq \widehat{\psi}_{\mathrm{FR}}$

C3 $\neg\,(\exists\, L \in \widehat{\psi}_{\mathrm{SH}}\, \exists\, P \subseteq \mathrm{SH}(\tilde{\mathrm{p}})\ \ L = \bigcup(N \in P)$
$\wedge\,(\exists\, v_1{,}v_2 \in \mathrm{FR}\ \ v_1{,}v_2 \in L \wedge \neg\, \exists\, N \in P\ \ v_1{,}v_2 \in N)$
$\wedge\,(\forall\, v \in \widehat{\beta}_{\mathrm{FR}}\, \forall\, N{,}M \in P\ \ (N \neq M \wedge v \in N) \rightarrow v \notin M))$

Here, $P$ is the set of sharing sets that $\tilde{\mathrm{p}}$ can join (thus they come from $\mathrm{SH}(\tilde{\mathrm{p}})$), $L$ is the sharing set in the answer abstract substitution resulting from the join, and the shared variables $v_1$ and $v_2$ are dependent in L but they are not dependent in any set of $P$. Furthermore, we ensure that the offending $P$ has at most one sharing set containing each free variable, since not two sets containing the same free variable can be active in one concrete substitution. Intuitively it can be seen that if $\neg$C3 holds, $\tilde{\mathrm{p}}$ can possibly bind the two independent shared variables.

Figure 10 gives an example which violates the condition C3, but now because it actually represents a situation where there is no non–strict independence. The abstract substitutions involved are $\widehat{\beta}$ = ([[XY] [YZ] [Y] [W]], [XZ]) and $\widehat{\psi}$ = ([[XYZW] [XY] [YZ] [Y]], $[XZ]$). The sets that fail the third condition are $L$ = [XYZW], $P$ = [[XY] [YZ] [W]], together with the variables $v_1$ = X and $v_2$ = Z. An example of corresponding concrete substitutions can be: $\beta$ = {X/A, Y/f([t(E, F)|A], B), Z/B, W/C} and $\psi$ = {X/A,

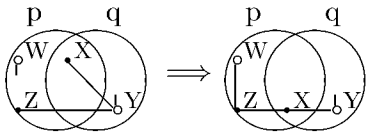



Figure 10: A situation without NSI where parallelism is avoided by condition C3.

$Y/f([t(E,F)|A],A)$, $Z/A$, $W/A\}$.

## 5 Minimal Run–Time Checks for Non–Strict Independence

In the previous section we have proposed conditions to be checked at compile–time in order to decide whether to run two goals in parallel. However, even if these conditions do not hold, we may yet try to execute them in parallel, provided that some a priori run–time checks succeed.

The purpose of the run–time checks is to ensure that goals will not be run in parallel when there is no non–strict independence, while allowing parallel execution in as many cases as possible when non–strict independence is present. This fact will be determined from the combination of compile–time analysis and the success of the run–time checks previous to the execution of the goals. Note that this is meaningful because the sharing component represents possible, not sure sharing sets.

In the previous section we proposed three conditions which had to hold for parallelization. Let us analyze what to do when each of the conditions is violated.

### 5.1 Condition C2 Violated $[\mathrm{FR} \not\subseteq \widehat{\psi}_{\mathrm{FR}}]$

If condition C2 does not hold, there is nothing to be done but realize that we cannot execute the goals in parallel (since the first goal possibly binds variables of the second, and no a priori check can avoid it).

### 5.2 Condition C1 Violated $[\exists\, L \in \mathrm{SH}\ \ \mathrm{FR}(L) = \emptyset]$

Let $\mathrm{SH}^-$ be the subset of SH consisting of the sets $L$ not obeying the above condition, and $\mathrm{SH}^+ = \mathrm{SH} - \mathrm{SH}^-$. For each of such $L$ a run–time check must be done in order to ensure that it is not active. Moreover, we must try to generate the least number of checks which covers every $L \in \mathrm{SH}^-$ without affecting any other sharing set (to preserve parallelism in valid situations).

The type of checks that can be used to prevent a sharing set $L$ from being active are listed below, with increasing complexity, along with the conditions that ensure that the valid situations are respected (assume that $\mathrm{X}, \mathrm{Y} \in L$).

- `ground(X)` if $\neg\exists\, N \in \widehat{\beta}_{\mathrm{SH}} - \mathrm{SH}^{-}\ \ \mathrm{X} \in N$
  where `ground(X)` is a predicate that is true if X is ground.
- `allvars(X,`$\mathcal{F}_{\mathrm{X}}$`)` if $\neg\exists\, N \in \widehat{\beta}_{\mathrm{SH}} - \mathrm{SH}^{-}\ \ \mathrm{X} \in N \wedge \mathrm{FR}(N) = \emptyset$ (this is always true if $\mathrm{X} \in \mathrm{var}(\tilde{\mathrm{p}}) \cap \mathrm{var}(\tilde{\mathrm{q}})$)
  where $\mathcal{F}_{\mathrm{X}} = \bigcup_{N \in \widehat{\beta}_{\mathrm{SH}},\, N \ni \mathrm{X}} \mathrm{FR}(N)$ and `allvars(X,S)` is a predicate that is true if every variable in X is in the list `S` (note that we can put FR instead of $\mathcal{F}_{\mathrm{X}}$, but since the later is smaller is more efficient).
- `indep(X,Y)` if $\neg\exists\, N \in \widehat{\beta}_{\mathrm{SH}} - \mathrm{SH}^{-}\ \ \mathrm{X},\mathrm{Y} \in N$
  where `indep(X,Y)` is a predicate that is true if X and Y do not share variables.
- `sharedvars(X,Y,`$\mathcal{F}_{\mathrm{X,Y}}$`)` if $\neg\exists\, N \in \widehat{\beta}_{\mathrm{SH}} - \mathrm{SH}^{-}\ \ \mathrm{X},\mathrm{Y} \in N \wedge \mathrm{FR}(N) = \emptyset$ (this is always true if $\mathrm{X} \in \mathrm{var}(\tilde{\mathrm{p}})$ and $\mathrm{Y} \in \mathrm{var}(\tilde{\mathrm{q}})$)
  where $\mathcal{F}_{\mathrm{X,Y}} = \bigcup_{N \in \widehat{\beta}_{\mathrm{SH}},\, N \ni \mathrm{X,Y}} \mathrm{FR}(N)$ and `sharedvars(X,Y,S)` is a predicate that is true if every variable shared by X and Y is in the list of variables S (also note that again we can put FR instead of $\mathcal{F}_{\mathrm{X,Y}}$)

### 5.3 Condition C3 Violated

$$
\begin{aligned}
[\exists\, L \in \widehat{\psi}_{\mathrm{SH}}\ \exists\, P \subseteq \mathrm{SH}(\tilde{\mathrm{p}})\ \ & L = \bigcup(N \in P) \\
\wedge\ & (\forall\, v \in \widehat{\beta}_{\mathrm{FR}}\ \forall\, N,M \in P\ \ (N \neq M\ \wedge\ v \in N) \rightarrow v \notin M) \\
\wedge\ & (\exists\, v_1,v_2 \in \mathrm{FR}\ \ v_1,v_2 \in L\ \wedge\ \neg\exists\, N \in P\ \ v_1,v_2 \in N)\ ]
\end{aligned}
$$

Once the checks for C1 have been computed, and taking into account only the sharing sets not rejected by these checks, the third condition is treated. Now, for each set of existential instances in the above formula, we have two conditions to be determined:

(P) Whether $v_1$ and $v_2$ can be dependent in a $\beta \in \gamma(\widehat{\beta})$ or not.
$\exists\, K \in \mathrm{SH}\ \ v_1,v_2 \in K$ ?

(Q) Whether $v_1$ and $v_2$ can be independent in a $\psi \in \gamma(\widehat{\psi})$ or not.
$\exists\, I, J \in \widehat{\psi}_{\mathrm{SH}}\ \ v_1 \in I \wedge v_2 \notin I \wedge v_2 \in J \wedge v_1 \notin J$ ?

Possible successes and failures of these two conditions yield the four cases below, listed along with the actions to be taken for them:

$\bar{P}\bar{Q}$ In this case, it is sure that $v_1$ and $v_2$ are independent before the execution of $\tilde{p}$ and dependent after it, so $\tilde{p}$ and $\tilde{q}$ cannot be parallelized.

$\bar{P}Q$ The second situation is when $v_1$ and $v_2$ are independent before $\tilde{p}$ and may or may not remain independent afterwards: we need a check that ensures that a sharing set in $P$ is impossible. These checks are handled in the same way as the ones for condition C1, minimizing their number.

$P\bar{Q}$ In this third case, we know that $v_1$ and $v_2$ are dependent after $\tilde{p}$, but it is not sure whether they were so before. Thus, the run–time check to cope with the eventuality is `dep(`$v_1$`,`$v_2$`)` (dependence check), that can be written in Prolog as $v_1$`==`$v_2$.

$PQ$ The last case is in a way the union of the two cases above: we have no sure knowledge about dependence of $v_1$ and $v_2$ neither in $\widehat{\beta}$ nor in $\widehat{\psi}$, so the check is the disjunction of the two previous checks.

For example, suppose we are trying to parallelize the goal "`p(X,Y,W,U),q(X,Y,U,Z)`" and the call and answer abstract substitutions are, respectively, $\widehat{\beta} = ([[X]\,[XY]\,[Z]\,[W]\,[ZW]\,[WU]], [YU])$ and $\widehat{\psi} = ([[XY]\,[Z]\,[WU]], [YU])$. Condition C1 is violated, being $\mathrm{SH}^{-} = [[X]\,[ZW]]$. So, the check for sharing set [X] would be `allvars(X,[Y])` (since `ground(X)` eliminates also [XY], which is legal), and the check for [ZW] would be `indep(Z,W)` (since `allvars(Z,` $\mathcal{F}_Z$`)` or `allvars(W,`$\mathcal{F}_W$`)` eliminate [Z] or [W], which are both legal). Condition C3 holds, so we are ready to parallelize the two goals, and the goal would be left as follows (here we omit the substitution of variables, to be explained in the next section):

```
(allvars(X,[Y]), indep(Z,W) -> p(X,Y,W,U) & q(X,Y,U,Z);
                               p(X,Y,W,U), q(X,Y,U,Z))
```

where "`A -> B; C`" is the prolog if-then-else construction and "`&`" is the (unconditional) parallel operator.

### 5.4 Run–Time Checks and Strict Independence

It is worth pointing out that if no information is obtained by analysis (or no analysis is performed, so abstract substitutions are $\top$) the run–time conditions computed by the method presented here are the conditions traditionally generated for strict independence (shared program variables ground, other program variables independent, see

[11] for more information). This is correct, since in absence of analysis information only strict independence is possible, and shows that this method is a strict generalization of the techniques which have been previously proposed for the detection of strict independence.

It can be easily shown how the tests reduce to those for strict independence: since there are no free variables in the call abstract substitution, the second and third conditions are met, but $\mathrm{SH}^- = \mathrm{SH}$. Thus, $\mathrm{SH}^-$ contains sharing sets containing a shared program variable (covered by a `ground/1` check on each) and sharing sets containing program variables of both goals (covered by a `indep/2` check on every pair).

For example, if we have a goal "`p(X,Y)&q(Y,Z)`" with $\widehat{\beta} = ([[\mathrm{X}]\,[\mathrm{Y}]\,[\mathrm{Z}]\,[\mathrm{XY}]\,[\mathrm{XZ}]\,[\mathrm{YZ}]\,[XYZ]], [])$ (i.e. $\top$, equivalent to no information), then $\mathrm{SH}^- = [[\mathrm{Y}]\,[\mathrm{XY}]\,[\mathrm{XZ}]\,[\mathrm{YZ}]\,[\mathrm{XYZ}]]$. The check `ground(Y)` covers all the offending sharing sets except [XZ], which is covered in turn by the check `indep(X,Z)`.

## 6 Renaming and Substituting Variables

In order to prevent partial answers of a branch that ultimately fail from pruning the search space of other goals, parallel goals are in principle run in independent environments (see [11, 12]). The standard solution for this problem is a run–time transformation of the goals to be executed in parallel. This transformation involves eliminating any shared variable among parallel goals by renaming or substituting all its occurrences so that no two occurrences in different goals remain the same, and adding some unification goals after the parallel conjunction to reestablish the lost links. Here we will attempt to perform this operation at least in part at compile–time, by defining a new predicate. Note that a mere renaming of variables at compile–time is not sufficient in general: we can have terms with shared variables inside.

The transformation procedure, for each goal, depends on what kind of variables occur in it (free or ordinary) and whether they are in sharing sets of SH or not. The different cases are listed below.

**a)** The simplest case is when the variables in the goal are not present in the sharing sets of SH. This means that this goal does not share variables with the others (is strictly independent) so no transformation is needed.

**b)** If the variables in the goal present in the sharing sets of SH are all known to be free (members of $\widehat{\beta}_{\mathrm{FR}}$), and no two are in the same sharing set, we simply rename

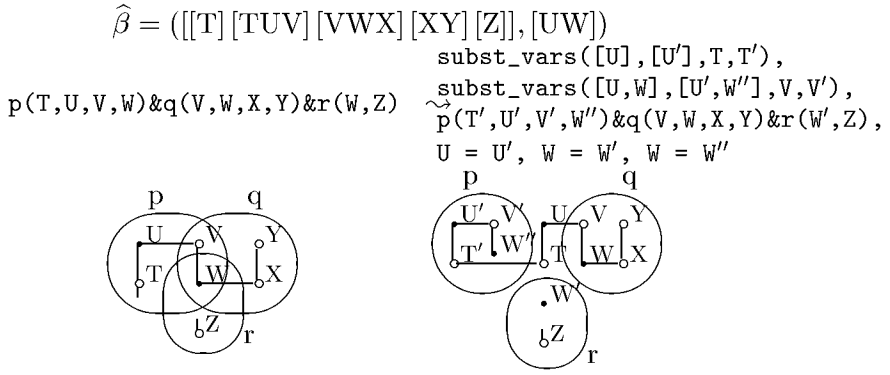



Figure 11: An example of variable substitution in a goal.

these variables and include the corresponding unification goals (or "back-binding" goals) after the parallel conjunction.

**c)** Else, a new approach has to be considered in order to deal with shared variables inside terms. Let us define the predicate `subst_vars`/4 with the following meaning:

`subst_vars(`$[X_1,\ldots,X_n]$`,`$[X'_1,\ldots,X'_n]$`,Z,Z')` :-
{`Z'` *is a term equal to* `Z` *but with the variables* $X_1,\ldots,X_n$ *substituted for the variables* $X'_1,\ldots,X'_n$ *respectively*}.

We add `subst_vars` for each variable in the goal present in the sharing sets of SH, substituting the free variables in the sharing sets with new ones. After the parallel conjunction we place the back-binding goals for the free variables renamed. Note that the isolation of the goals could be achieved with the `copy_term`/2 predicate, but in a less efficient way, since this predicate copies everything except, perhaps, the ground structures of the term, and `subst_vars` can also save the copying of non–ground structures containing variables which do not need to be renamed. Furthermore, with `copy_term` we need to unify the entire structures after the parallel call, but with `subst_vars` we only need to unify the renamed free variables.

The three cases listed are tried in order, the transformation being performed in steps, one goal at a time. Figure 11 illustrates an example transformation, representing in pictures the possible sharing patterns.

## 7 Example Parallelization of a Program

As an example, in this section, we will show how to apply the proposed methods to a concrete program (quicksort using difference lists) in order to achieve non–strict independence. Although the program is small, we think that it is of sufficient entity to show the potential of the proposal, and at the same time allows to present all the process of the parallelization.

The quicksort program we will use follows, with the abstract substitutions obtained by the analyzer annotated at each point of the program:

```
qsort(I,O) :-                              %[[O]],[O]
        qsort(I,O,[]).                     %[],[]
qsort([],L,L).
qsort([X|Xs],L,L2) :-                      %[[L],[L2],[Left],[Right],[L1]],[L,Left,Right,L1]
        partition(Xs,X,Left,Right),        %[[L],[L2],[L1]],[L,L1]
        qsort(Left,L,[X|L1]),              %[[L,L1],[L2]],[L1]
        qsort(Right,L1,L2).                %[[L,L2,L1]],[]
partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right) :-      %[[Right],[Left1]],[Right,Left1]
        E<C,                               %[[Right],[Left1]],[Right,Left1]
        !,
        partition(R,C,Left1,Right).        %[],[]
partition([E|R],C,Left,[E|Right1]) :-      %[[Left],[Right1]],[Left,Right1]
        E>=C,                              %[[Left],[Right1]],[Left,Right1]
        !,
        partition(R,C,Left,Right1).        %[],[]
```

We will concentrate on the parallelization of the `qsort/3` predicate, by first analyzing whether it is possible to parallelize the first and second goal of the recursive clause of `qsort/3`. We have $\tilde{p}$ = `partition(Xs,X,Left,Right)`, $\tilde{q}$ = `qsort(Left,L,[X|L1])`, $\widehat{\beta}$ = ([[L] [L2] [Left] [Right] [L1]], [L Left Right L1]) and $\widehat{\psi}$ = ([[L] [L2] [L1]], [L L1]). Then, we compute the sets SH = [[Left]] and FR = [Left]. But condition C2 is not met, since "Left" is not in $\widehat{\psi}_{\mathrm{FR}}$, so the goals are not non–strictly independent. In a similar manner it can be shown that the first and third goal of the clause are not non–strictly independent too.

Finally, let us try with the second and third goals in the same clause. Now $\tilde{p}$ = `qsort(Left,L,[X|L1])`, $\tilde{q}$ = `qsort(Right,L1,L2)`, $\widehat{\beta}$ = ([[L] [L2] [L1]], [L L1]) and $\widehat{\psi}$ = ([[L L1] [L2]], [L1]). The computed sets are SH = [[L1]] and FR = [L1]. But now the conditions hold: FR([L1]) = [L1] $\neq \emptyset$, FR = [L1] $\subseteq \widehat{\psi}_{\mathrm{FR}}$ = [L1] and there does not exist sharing sets meeting ¬C3. So in this case we have non–strict independence, and

no run–time checks are needed (note also that the goals are not strictly independent, since they share the variable "L1").

The last step is to see whether we need to rename or substitute any variable in the goals. If we revise the possible cases in section 6, we can see that we are in the "b" case. So we only need to rename the variable "L1", and the predicate `qsort/3` would be left as:

```
qsort([],L,L).
qsort([X|Xs],L,L2):-
        partition(Xs,X,Left,Right),
        qsort(Left,L,[X|L1]) & qsort(Right,L1_prime,L2),
        L1=L1_prime.
```

We hope that this example has shown how the notion of non–strict independence can allow more parallelism than strict independence, especially when dealing with difference structures or structures containing variables (the program cannot be parallelized with strict independence, except if the renaming is explicitly coded by the programmer).

## 8 Towards an Improved Analysis for Non–Strict Independence

Although, in general, a more precise analysis is not always necessarily a better analysis (because more accurate information requires more time), it is certain that for different purposes we want different pieces of information and that the analysis used so far can be improved.

In the case of our study, we think that the key idea is to have access to the greatest number of run–time free variables, in order to check its possible instantiations, having at the same time the more accurate information about the sharing. To achieve this goal, sharing and freeness could be combined with other analysis, like linearity analysis [23], a depth-k approach [21], or with a recursive type analysis, mainly for lists, to deal, for example, with lists of free variables (see [2, 15, 1]). All these alternatives will be taken into account in further work.

## 9 Conclusions

Despite the advantage of "non–strict" independence (NSI) over "strict" independence (SI) in terms of generality and the amount of parallelism it can exploit, all compilation technology developed to date has been based on SI, presumably due to the complex-

ity of compile–time detection of NSI. In an attempt to fill this gap we have presented several techniques for achieving this compile–time detection. The proposed techniques are based on the availability of certain information about run–time instantiations of program variables —sharing and freeness— for which compile–time technology is available, and for the inference of which new approaches are being currently proposed. We have also presented techniques for combined compile–time/run–time detection of NSI, proposing new kinds of run–time checks for this type of parallelism as well as the algorithms for implementing such checks. We have presented an efficient algorithm for performing combined compile–time/run–time renaming of variables to ensure that non–strictly independent goals run in separate environments with respect to their shared variables. Finally, an example of the application of the proposed methods to a concrete program has been given.

We are in the process of implementing these algorithms to interfacing them with the sharing+freeness analyzer implementation available to us with the objective of obtaining a complete compile–time parallelizer capable of detecting NSI. We are also planning on looking, in the light of the techniques developed, at other types of analyses which may provide additional information useful for such parallelization.

## References


1. A. Bansal and L. Sterling. An Abstract Interpretation Scheme for Identifying Inherent Parallelism in Logic Programs. *New Generation Computing*, (7):273–324, 1990.
2. M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seattle, Washington, August 1988. MIT Press.
3. Michael Codish, Dennis Dams, Gilberto File, and Maurice Bruynooghe. Freeness Analysis for Logic Programs - And Correctness? In *Proc. Int'l. Conf. on Logic Programming*. MIT Press, 1993. To appear.
4. J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California At Irvine, 1983. Technical Report 204.
5. A. Cortesi and G. File. Abstract Interpretation of Logic Programs: an Abstract Domain for Groundness, Sharing, Freeness and Compoundness Analysis. In *ACM Symposium on Partial Evaluation and Semantic Based Program Manipulation*, pages 52–61, New York, 1991.
6. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2 and 3):103–179, July 1992.
7. D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.
8. G. Gupta and B. Jayaraman. Compiled And-Or Parallelism on Shared Memory Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 332–349. MIT Press, October 1989.
9. S. Haridi and S. Janson. Kernel Andorra Prolog and its Computation Model. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, June 1990.
10. M. Hermenegildo and K. Greene. The &-prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
11. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.
12. M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.
13. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

14. Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

15. G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

16. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

17. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *1990 International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

18. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

19. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, July 1992.

20. B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

21. T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

22. K. Shen. *Studies in And/Or Parallelism in Prolog.* PhD thesis, U. of Cambridge, 1992.

23. H. Sondergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

24. R. Sundarajan. An Abstract Interpretation Scheme for Groundness, Freeness, and Sharing Analysis of Logic Programs. Technical Report CIS-TR-91-06, U. of Oregon, Eugene, Oregon 97403, October 1991.

25. M. Bruynooghe V. Dumortier, G. Janssens and M. Codish. Freeness Analysis in the Presence of Numerical Constraints. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 100–115, Budapest, Hungary, June 1993. MIT Press.

26. D. H. D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.

27. H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR-Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.